

Verification of C++ Flight Software with the MCP Model Checker

S. Thompson, G. Brat
USRA/RIACS
NASA Ames Research Center
MS 269/2
Moffett Field, CA 94035-1000
650-604-{0456,1105}
{thompson,brat}@email.arc.nasa.gov

Abstract—The Constellation project at NASA calls for designing a Crew Exploration Vehicle (Orion, also called CEV) and Cargo Launch Vehicle (Ares, also called CLV). Both projects will rely on newly designed flight control software. The verification of these C++ flight codes is critical, especially for Orion, since human life will be at stake. There exist some commercial tools for the verification of C++ code. However, none of the commercially available tools does a good job a finding bugs dealing with concurrency. Yet both software for Orion and Ares are expected to be multi-threaded. With this work we are proposing to address the issue by developing a suite of tools that can be used to verify C++ code. Our tools will range from a static analyzer (based on abstract interpretation like C Global Surveyor) to a model checker (MCP, which we present in this paper) including a symbolic execution engine for test case generation (TPGEN). This paper focuses on MCP and its application to Aerospace software.^{1,2}

There exist some commercial tools for the verification of C++ code. Most of them are focusing on testing (test drivers, test cases), and, some are using more advanced techniques such as static analysis (Coverity, Klocwork, PolySpace, Code Sonar to name only a few). However, none of the commercially available tools does a good job a finding bugs dealing with concurrency. Yet both software for Orion and Ares are expected to be multi-threaded.

Concurrency errors are usually addressed by model checking tools. Unfortunately, available model checkers either deal with Java (like Java Path Finder, JPF [1]) or require a translation to a specialized modeling language (e.g., to Promela for the SPIN model checker [2]). There exist some ways of partially addressing the problem: for example, model-driven verification allows C code fragments to be embedded in Promela models. However, there does not exist any tool capable of model checking C++ code directly.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. VERIFYING C++ CODE	1
3. THE MCP MODEL CHECKER	4
4. THE OAE CASE STUDY	5
5. RELATED WORK	6
8. CONCLUSIONS	7
REFERENCES	7
BIOGRAPHY	8

1. INTRODUCTION

The Constellation project at NASA calls for designing a Crew Exploration Vehicle (Orion, also called CEV) and Cargo Launch Vehicle (Ares, also called CLV). Both projects will rely on newly designed flight control software. In the case of Orion, the software process relies on a model-based approach, in which C++ software will be generated from high-level models (Statecharts or a similar formalism). The software design of Ares also calls for a C++ implementation. The verification of these C++ flight codes is critical, especially for Orion since human life will be at stake.

With this work we are proposing to address the issue by developing a suite of tools that can be used to verify C++ code. Our tools will range from a static analyzer (based on abstract interpretation like C Global Surveyor) to a model checker (MCP, which we present in this paper) including a symbolic execution engine for test case generation (TPGEN). This paper focuses on MCP [3] and its application to Aerospace software. For this paper, we chose a small example called OAE, On-Board Abort Executive, which implements the monitoring and execution of abort conditions during launch. In the future, we will also apply MCP to the verification of International Space Station (ISS) software such as the Water Recovery System (WRS) and the Urine Processing Assembly (UPA).

This paper is organized as follows. The first section describes our global approach to the verification of C++ code. We briefly describe the V&V technologies we are developing and how they fit together. The second section focuses on MCP and its capabilities. The third section then details our case study, i.e., applying MCP to the OAE code. We then discuss related work and present our conclusions.

2. VERIFYING C++ CODE

C++ is a very complex language in terms of verification. C++ relies on powerful (from a coding point of view) but

¹ 1-4244-1488-1/08/\$25.00 ©2008 IEEE.

² IEEEAC paper #1177, Version 1, Updated October 19, 2007

complex (from a verification point of view) constructs such as

- templates, which are addressed through their instances (rather than the generic form) in verification,
- dynamic object allocation, which abundant use in C++ code renders static analysis difficult,
- implicit (and explicit) constructors and destructors can have confusing behaviors,
- multiple inheritance also has perverse effects, and,
- overloaded operators can also be tricky.

This multitude of features is even more difficult to verify when they are combined (as is routinely done in C++ programs).

This complexity is a powerful motivation for having tools that can assist in the verification of C++ programs. It is also a source of problems when designing such tools. When attempting to define a verification strategy for C++, it quickly becomes clear that there does not exist a “silver bullet” tool which can do the job. In our case, we rapidly decided that developing a suite of complementary tools would be the right solution.

To address C++, we built on the experience gathered while analyzing C programs. In [4,5], we showed how we used abstract interpretation techniques to build a static analyzer for flight software written in C. We applied CGS to flight code for Mars missions, controllers for the ISS, and even, to the main engine controller for the Space Shuttle. We were quite happy with the results of the tool.

However, we realized that we would not be able to handle all (even a large subset) classes of errors with sufficient precision. Indeed, static analyzers almost always generate warnings that might correspond to real errors or merely unfeasible by-products of the approximations done by the tool for scalability reasons (also commonly called false positives). Classifying a warning as a real error or as a false positive takes time. Therefore, a tool that generates a high rate of warnings is likely to be ignored by the user community. So, the solution is to use other tools with different characteristics in a process in which each tool compensates for the weaknesses of another tool.

In our case we have selected three types of verification technology, which we believe complement each other:

1. static analysis, which catches most of the so-called runtime errors,
2. model checking, which can catch many concurrency errors, and,

3. symbolic execution, which can generate automatically test cases that exercise precise paths in a program.

Static analysis

The goal of static analysis is to assess code properties without executing the code. Several techniques can be used to perform static analysis, such as theorem proving, data flow analysis [6], constraint solving [7], and abstract interpretation [8]. The purpose of most static analyzers is to catch runtime errors.

Runtime errors are errors that cause exceptions at runtime. Typically, in C, either they result in creating core files or they cause data corruption that may cause crashes. In this study we mostly looked for the following runtime errors:

- Access to un-initialized variables
- Access to un-initialized pointers
- Out-of-bound array access
- Arithmetic underflow/overflow
- Invalid arithmetic operations (e.g., dividing by zero or taking the square root of a negative number)
- Non-terminating loops
- Non-terminating calls.

The price to pay for exhaustive coverage is incompleteness: the analyzer can raise false alarms on some operations that are actually safe. However, if the analyzer deems an operation safe, then this property holds for all possible execution paths. The program analyzer can also detect certain runtime errors which occur every time the execution reaches some point in the program. Therefore, a program analyzer can be used either as a debugger that detects runtime errors statically without executing the program or as a preprocessor that reduces the number of potentially dangerous operations that have to be checked by another validation process (code reviewing, test writing, and so on).

Model checking

Originally model checking was a method to formally verify finite state systems [9]. This is achieved by verifying if the model (derived from the design phase or by abstraction of the code) satisfies a logical property (derived from the requirements). Properties are often expressed as temporal logic formulas, but simple assertions can also be checked.

Explicit state model checking (which is what we are using in this work) uses an explicit representation of the system's global state graph, usually given by a state transition function. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure, and property validation amounts to a partial or complete exploration of the state space.

The first interesting aspect is how to represent the state space and its transition. In SPIN, the Promela language is used to specify the model; it therefore requires a translation pass to check software programs. However, several approaches have been designed recently to make explicit-state model checking work directly on code. For example, JPF is a model checker that works on Java bytecode. In JPF, the bytecodes of the program describe the state transitions, the state being defined by the actual state of the program at each program point (or by an abstraction of this state). MCP is similar to JPF in the sense that it works directly on software, but it uses a process similar to SPIN.

The second characteristic of explicit-state model checking is that it explores systematically every reachable state from the root. If one considers the example of a sequential program, this corresponds to a trace through the system. If one considers a multi-threaded program, then it corresponds to exploring all the possible interleavings of the threads in the program. Traditionally, the search follows a DFS pattern. However, there have been attempts at using different search strategies, including BFS and some heuristics.

Lastly, linear temporal logic properties are the properties most often checked by explicit-state model checkers. Temporal logic properties express temporal properties over an execution trace. Temporal operator can express the fact that

- something always happens in a trace,
- something happens next in a trace, and,
- something will eventually happens in the trace.

This allows a user to express properties such as “Always after p there is eventually q ”. This particular example is interesting since it corresponds to a response property, which is often used in specification for embedded systems (especially in control). In practice, users have problems expressing specification in LTL. Most often explicit-state model checkers are used to verify assertions (at some program point) and problems due to multi-threading (deadlocks, race conditions, and so on).

Symbolic execution

In the symbolic execution of programs, instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values [10]. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. Moreover, the symbolic execution engine generates and stores constraints that satisfy conditional branch type statements. By solving these constraints, one can decide if the path being explored is feasible. Once this is decided, the constraint gathered for inputs can be solved to generate test cases that will exercise the code at run time.

The major difference between symbolic execution and model checking is the fact that symbolic execution manipulates

symbolic names instead of real values. A model checker tries different values, but it manipulates concrete values. However, both techniques tend to take a path-sensitive approach to state space exploration.

Now, symbolic execution is actually closer to static analysis. It is actually not clear where static analysis stops and symbolic execution starts. However, static analysis tends to propagate abstractions of possible values while symbolic execution propagates a system of constraints that, if solved, represents the exact values that can be taken by the program. The static analysis abstractions are safe approximations of the constraints generated by the symbolic execution engine. The difference stems from the fact that a static analyzer aims at exploring a whole state space while symbolic execution generally focuses on a given path in the state space.

For example, JPF offers a symbolic execution capability to generate test data; it uses a BytecodeFactory to override JPF's core bytecodes to generate concrete test cases. In a nutshell, this works by using the JPF field/stackframe attribute system to collect symbolic path conditions, which are then fed into a constraint solver to obtain concrete test data.

We have already developed a symbolic execution engine for C++, but it relies on an earlier front-end of LLVM (LLVM 1.8). As a result it does not handle all C++ programs yet. We are in the process of porting it to LLVM 2.1.

Tool Interactions

The model checker can benefit from the static analyzer in several ways. First, the static analyzer might pinpoint regions of the state space that need further exploration by the model checker. Second, static analysis can be used to provide partial order reduction information, which allows a model checker to ignore some interleavings, thus reducing the state space being explored without missing behaviors.

The static analysis can benefit from the model checker in two ways. First, as shown in [11], the model checker can and the static analyzer can work in symbiosis. The model checker can provide precise alias information that can be used by the static analyzer to refine its partial order reduction analysis. Second, the model checker can be used to refine the precision of the static analysis by exploring systematically regions of the state space with warnings. It can also be used to provide full counter-examples corresponding to errors (instead of the plain source line information returned by the static analyzer).

The symbolic execution engine is useful to both the model checker and the static analyzer because of its ability to compute test cases (by solving constraints on inputs). This feature can be used to turn static analyzer warnings into real errors (when they're not false positives) by providing a test case that will trigger the error. It can also be used to close the environment for the model checker. The goal is to provide (input) drivers for the model checker to feed concrete value into the program being model checked.

3. THE MCP MODEL CHECKER

This section focuses on MCP and its capabilities. We also briefly describe the LLVM framework [12] since our tools are built on top of it.

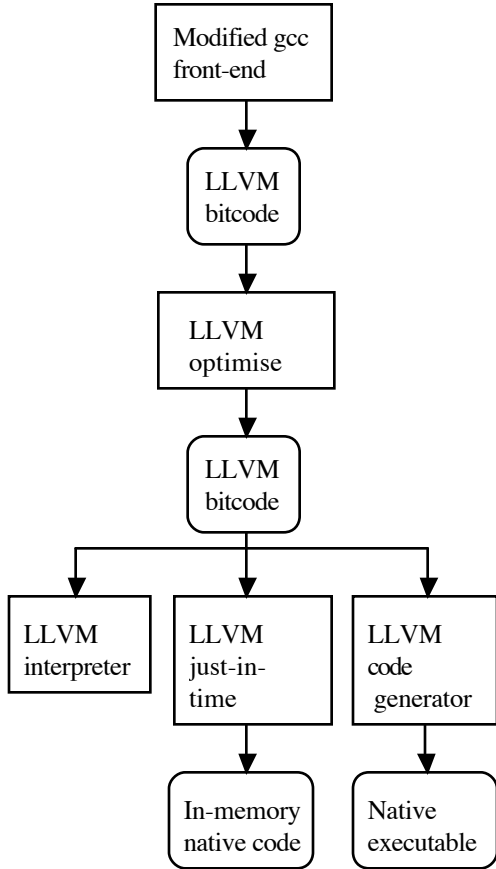


Figure 1. Simplified LLVM Architecture.

The LLVM framework

The MCP model checker is built on top of the LLVM framework. Actually all of our C/C++ analysis tools are using the LLVM framework. Figure 1 shows a simplified version of the LLVM flow. Note that many LLVM tools have been omitted here for clarity. LLVM is a large, rich toolset, so we concentrate on the subsystems that are specifically relevant to MCP. A modified version of the gcc front-end is used to parse the C++ source code and to lower most of the language's constructs to a level closer to that of a typical C program. The original gcc back-end is discarded in favor of emitting LLVM bytecode, which is then optimized and passed on to various alternative back-end.

One of the main motivating ideas behind LLVM was enabling whole program optimization, something that has been traditionally difficult to approach with existing compiler architectures. The LLVM bytecode format has been specifically designed to support this and other kinds of optimization. A Static Single Assignment (SSA) representation is adopted, making many analyses and

transformations (including ours) far more straightforward than they might otherwise be.

The MCP Architecture

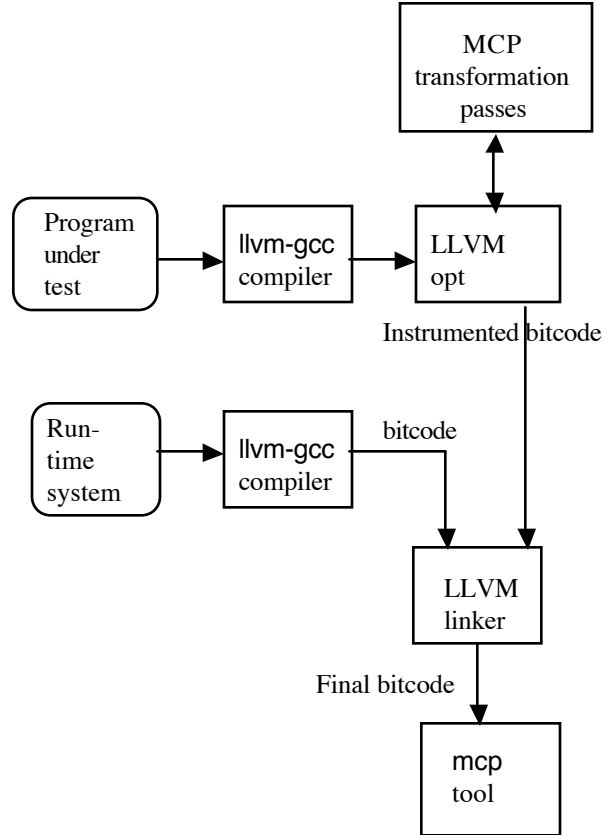


Figure 2. MCP Architecture.

MCP has a simple, expandable architecture (See Figure 2). We refer the reader to [3] for a detailed description of the tool. We now give an overview of MCP's main elements.

Transformation passes help transform the program under test into a version that can model check itself. They are implemented as extension modules for LLVM's opt program transformation framework. Transformations provide

- instrumentation for tracing: model checkers produce counter-examples (or traces) when they find violations
- Memory read/write instrumentation, which is needed for backtracking efficiently. Backtracking is done the end of the current path being explored or when a violation has been found.
- Yield point insertion, which is needed to explore all interleavings due to thread scheduling or switch between different values of a data abstraction.

The run-time system, which is implemented in C++, is compiled with LLVM's gcc front-end and then linked with the program under test after it has been transformed. Its primary purpose is to intercept system calls that MCP needs

to handle differently, e.g. `printf`, `malloc`, `free`, `memset`, `memmove` and `memcpy`. This approach also provides a convenient place to implement compatibility wrappers that allow code written to specific operating system APIs to be handled without modification.

From a practical point of view, model checking is initiated by users by running the `mcp` command-line application. The `mcp` tool comprises an instance of the LLVM just-in-time (JIT) compiler environment, as well as MCP's implementations of memory versioning, hashing, state space searching, etc.

Searching for assertions

The run-time system intercepts calls to the commonly used `assert` C/C++ library function and passes them on to the model checker core as calls to `mcp_assert`, which has the effect of halting execution and generating a backtrace when an assertion failure is detected. Currently, this is the primary means by which failure conditions are identified in programs, though support for more complex conditions expressed in the form of temporal logic expressions is planned.

In searching a program's state space for failure conditions, explicit-state model checkers must essentially search a decision tree that represents the choices that are made that influence a program's run-time functionality. Normally, for reasons of practicality, a test driver is implemented that concentrates the attention of the model checker on a specific part of the state space.

In most cases, numeric decisions should be implemented in the driver (as in our OAE case study). A numerical decision is normally requested by the test driver for a program, e.g.

```
int a;
```

```
mcp_int_decision(1, 5, &a);
```

In the above example, the `mcp_int_decision` call effectively splits reality into five separate versions, with `a` taking a distinct value from the range `[1, 5]` in each case.

MCP implements several search strategies that may be chosen independently of other options. MCP offers the traditional breadth-first and depth-first search strategies as well as

- Heuristic search: in this mode, MCP's normal double-ended state queue is replaced by a priority queue that returns states in descending order of a program-specified metric.
- Randomized search, which is broadly similar to breadth-first search, with the exception that next-states are inserted at randomly chosen positions within the state queue. This search strategy should not be regarded as equivalent to conventional randomized testing. In

practice, randomized search has a 'look and feel' somewhere between that of breadth- and depth-first search, but in practice it has less tendency to get stuck in local minima.

- Interpret-only: since programs annotated with calls to the MCP API can not easily be executed natively without modification, this mode provides a convenient means of executing programs conventionally within the MCP environment. As usual, however, all visited instructions and source lines are logged, allowing a counterexample trace to be generated if a fault is encountered.

4. THE OAE CASE STUDY

In previous experiments in the RSE group at NASA Ames Research Center, the JPF model checker (especially, its symbolic execution capabilities) was applied to a Java translation of On-board Abort Execution (OAE). The experiment successfully demonstrated that these verification technologies can be applied to real NASA code. However, our goal is to provide technologies that can directly apply to C and C++ code. Therefore, we are running again the same experiment using MCP, our model checker for C++.

The OAE software is a relatively small piece of software which has been developed to study abort conditions for the Orion and Ares vehicles in the Constellation program. It is part of the ANTARES simulation environment (ANTARES is derived from the ARES simulation environment developed for the Space Shuttle). Included in ANTARES is code for simulating flight dynamics, which includes off-nominal dynamics up to a point.

The first step consists of annotating the OAE program with assertions. We basically added an assertion each time an abort was raised in the code. Violating an assertion therefore means that an abort is not raised as it should.

```
int main(int argc, char **argv)
{
    FLIGHT_RULE_CHECK fr_chk;
    SCENARIO_EXEC scen_exec;

    // Zero the data structures (this gives us
    // determinism without needing us to
    // manually set everything)
    memset(&fr_chk, 0, sizeof(fr_chk));
    memset(&scen_exec, 0, sizeof(scen_exec));

    // Enable the flight rule checker
    fr_chk.enable = On;

    // Conditions for FR_A_2_A_1_A
    int t1;
    mcp_int_decision(1, 3, &t1);
```

```

fr_chk.inputs_fr.sysIN_fr.stage1_chmbr_pres = t1;
fr_chk.iloads_fr.fr_stage1_chmbr_pres_min = 1.5;
fr_chk.iloads_fr.fr_stage1_chmbr_pres_max = 2.5;

asc_oae_fr_check(&fr_chk, &scen_exec);

if(fr_chk.work_fr.abort_now_flag == On)
{
    printf("$$$ Abort: %s\n",
        fr_chk.outputs_fr.fr_err_msg);
}
else
{
    printf("$$$ No abort.\n");
}

// Stop when we encounter any abort condition
mcp_assert(fr_chk.work_fr.abort_now_flag == Off);

return 0;
}

```

Figure 3. Driver for model checking OAE.

The second step in the process of applying MCP to OAE is to build a driver for OAE (which usually sits in the ANTARES environment). The goal is to design a driver that can exercise all realistic numerical inputs to OAE. The simplified version of the driver shown in Figure 3 illustrates how the data structures taken as inputs by OAE are initialized and how the main OAE function is called. Remember that MCP has already transformed the OAE software into a version that actually model checks itself (instead of simply executing).

The test driver chooses a value for the Stage 1 chamber pressure from a range of values; then it uses either a depth- or breadth-first search to find a value that causes an abort to be signaled.

```

### System stack = 0x00000000, Initial stack =
0x00129000, Scheduler Stack = 0x00125000

--- $$$ No abort.
--- $$$ Abort: FR A_2_A_1_A: stage1 engine
chamber pressure limit exceeded

### ascent_oae_fr_check.cpp(76): Assertion
failure: fr_chk.work_fr.abort_now_flag == Off
@@@ Numeric decision: 0 -> 0x00000000
...   %argc_addr = alloca i32          ; <i32*>
[#uses=2]
...   %argv_addr = alloca i8**         ;
<i8***> [#uses=2]
...   %retval = alloca i32, align 4    ;
<i32*> [#uses=4]
...   %tmp = alloca i32, align 4       ; <i32*>
[#uses=4]
...   %fr_chk = alloca
%struct.FLIGHT_RULE_CHECK, align 16
...   %scen_exec = alloca
%struct.SCENARIO_EXEC, align 16
...   %t1 = alloca i32, align 4        ; <i32*>
[#uses=4]
...   "alloca point" = bitcast i32 0 to i32
; <i32> [#uses=0]

```

```

+++ ascent_oae_fr_check.cpp (41): int main(int
argc, char **argv)
...   %argc = bitcast i32* %argc_addr to { }*
; <{ }*> [#uses=1]
...   store i32 %argc1, i32* %argc_addr
-----
--
<snipped for brevity>
-----
--
@@@ Numeric decision: 1 -> 0x0012a8c0
-----
--
<snipped for brevity>
-----
Currently executing block: -
-----
@@@ Thread decision: 165870272
...   %tmp36 = icmp ne i32 %tmp35, 0      ;
<i1> [#uses=1]
...   %tmp3637 = zext i1 %tmp36 to i8      ;
<i8> [#uses=1]
...   %toBool38 = icmp ne i8 %tmp3637, 0
; <i1> [#uses=1]
...   br i1 %toBool38, label %cond_true39,
label %cond_next43
+++ ascent_oae_fr_check.cpp (76):
mcp_assert(fr_chk.work_fr.abort_now_flag ==
Off);
...   %tmp40 = getelementptr [24 x i8]*
@.str804, i32 0, i32 0 ; <i8*> [#uses=1]
...   %tmp41 = getelementptr [18 x i8]*
@.str805, i32 0, i32 0 ; <i8*> [#uses=1]
...   %tmp42 = getelementptr [37 x i8]*
@.str806, i32 0, i32 0 ; <i8*> [#uses=1]
...   call void @mcp_stop( i8* %tmp40, i32 76,
i8* %tmp41, i8* %tmp42 )
-----
--

```

Figure 4. Trace resulting from model checking OAE.

The third step consists of model checking the program. As a result of the model checking process, we obtain the trace shown in Figure 4. Notice that in the first path explored by the model checker, no abort was found and therefore no assertion was triggered. This corresponds to the line:

--- \$\$\$ No abort.

The model checker then backtracked explored another path in which an assertion was triggered as shown by the following line:

--- \$\$\$ Abort: FR A_2_A_1_A: stage1 engine chamber pressure limit exceeded

At this point, the model checker stops its exploration, backtracks and dumps the path that led to the failed assertion.

5. RELATED WORK

This section describes work related to model checking, especially, software model checking. The work cited here may not have been applied to Aerospace software.

Structurally, MCP probably bears closest resemblance to JPF [1], though at the time of writing it does not approach JPF's maturity. The most significant differences between JPF and MCP stem from the differences between Java and C++; for example, JPF takes advantage of reflection and the standard threading package in Java, which MCP can not since those features are not present in C or C++.

Since JPF is a explicit-state model checker "a la SPIN", MCP is also a close cousin to SPIN [2]. Besides the explicit-state model, MCP also shares with SPIN the concept of compiling the model checking problem into an executable program. SPIN starts with Promela models while MCP performs transformations on C/C++ programs to embed the model checking problem into the original program.

Another model checker directly addressing C++ is Verisoft [13], which takes a completely different approach. Verisoft follows a stateless approach to model checking while MCP follows a conventional explicit-state model similar to SPIN [2].

CBMC is a bounded Model Checker for C and C++ programs [14]. It can check properties such as buffer overflows, pointer safety, exceptions and user-specified assertions. CBMC does model checking by unwinding loops and transforming instructions into equations that are passed to a SAT solver. Paths are explored only up to a certain depth.

There are, however, several model checkers that address C. SLAM [15] is really more of a static analyzer than a model checker. It relies heavily on abstractions, starting from a highly abstracted form and building up to a form that allows a complete analysis.

CMC [16] uses an explicit-state approach, but it requires some manual adaptation when dealing with complex types (*pickle* and *unpickle* functions).

Finally, there have been some attempts within NASA to use the Valgrind tool [17] as a crude model checker. Unfortunately, it implies using very crude steps between transitions.

Other approaches to model checking code involve a translation step, be it automatic or manual. For example, Bandera [18] provides model checking of Java programs by translating automatically the program into a Promela [2], PVS [19] or SMV [9] model.

8. CONCLUSIONS

In this paper we describe our approach to the verification of C++ code for aerospace applications. We are building on top of the LLVM framework a suite of tools including a static analyzer, a symbolic execution engine, and a model checker which can support the full C++ language. We have

described how each of these tools address specific error categories (runtime for static analysis, concurrency for model checking, and general error classes for symbolic execution) as well as how each tool can interact with other tools to improve the verification results.

We have partial versions of all the tools, but we have just completed a practical version of the model checker, MCP. We briefly describe its design and show how it can be applied to the verification of the On-board Abort Executive (OAE) prototype for the Orion and Ares projects. Using assertions, we are verifying that each abort condition is indeed addressed by the prototype.

Our future plans include porting the symbolic execution engine from LLVM 1.9 to LLVM 2.1 and use it to do automatic test case generation for OAE and other aerospace C++ software. We also intend to port CGS (the static analyzer for C) to the LLVM framework so that we can offer static analysis of C++ programs.

REFERENCES

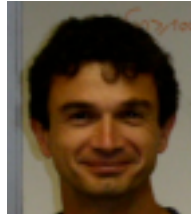
- [1] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda "Model Checking Programs." *Automated Software Engineering Journal*, volume 10, number 2, April 2003.
- [2] G.J. Holzmann, "The SPIN Model Checker", Addison-Wesley, 2004.
- [3] S. Thompson, G. Brat, "The MCP Model Checker," submitted to PEPM'08.
- [4] A. Venet and G. Brat, "Precise and Efficient Static Array Bound Checking for Large Embedded C Programs," *International Conference on Programming Language Design and Implementation Proceedings*, 231–242, 2004.
- [5] G. Brat and A. Venet, "Precise and scalable static program analysis of NASA flight software". In *Proceedings of the 2005 IEEE Aerospace Conference*, Big Sky, MT, 2005.
- [6] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley. 1986.
- [7] A. Aiken and M. Fähndrich, "Program Analysis using Mixed Term and Set Constraints," 4th *International Static Analysis Symposium Proceedings*, 1997.

- [8] P. Cousot and R. Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," 4th Symposium on Principles of Programming Languages Proceedings, 238–353, 1977.
- [9] E.M. Clarke, O. Grumberg, and, D.A. Peled, "Model Checking". MIT Press, 1999.
- [10] S. Anand, C. Pasareanu, W. Visser, "Symbolic Execution with Abstract Subsumption Checking," Proc. of 13th International SPIN Workshop on Model Checking of Software (SPIN), 2006.
- [11] G. Brat and W. Visser, "Combining Static Analysis and Model Checking for Software Analysis," Proceedings of ASE2001. San Diego, November 2001.
- [12] LLVM web page: <http://llvm.org/>
- [13] P. Godefroid, "VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software," Proceedings of the 9th Conference on Computer Aided Verification, Haifa, June 1997. Lecture Notes in Computer Science, vol. 1254, pages 476–479, Springer-Verlag.
- [14] E. Clarke, D. Kroening, F. Lerda, "A Tool for Checking ANSI-C Programs," Proc. of TACAS'04, pages 168–176, 2004.
- [15] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, A. Ustuner, "Thorough static analysis of device drivers," Proc. EuroSys'06 (European Systems Conference), 2006.
- [16] M. Musuvathi, A. Chou, D. L. Dill, D. Engler, "Model checking system software with CMC," Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC, pp. 219–222, 2002.
- [17] J. Seward, N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, 2005.
- [18] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code," Proceedings of the 22nd International Conference on Software Engineering, 2000.
- [19] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. K. Srivas, "PVS: Combining Specification, Proof Checking, and Model Checking," CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, pp. 411–414, 1996.

Dr. Sarah Thompson received a PhD from Cambridge University in 2006. She has a background in commercial software development and hardware design. Her PhD research applied abstract interpretation and program transformation techniques to electronic design for high radiation environments. She moved to NASA Ames in July 2006, and now specializes in model checking and static analysis of flight software.



Dr. Brat received his Ph.D. in Electrical & Computer Engineering in 1998 (The University of Texas at Austin, USA). He has specialized on the application of static analysis to software verification. From 1997 to June 1999, he worked at MCC where he led a project that developed static analysis tools for software verification. In June 1999, he joined the Automated Software Engineering group at the NASA Ames Research Center and focused on the application of static analysis to the verification of large software systems. He co-developed and applied static analysis tools based on abstract interpretation to the verification of software for Mars missions at JPL, various ISS controllers at MSFC, and the ISS Biological Research Project at the NASA Ames Research Center.



BIOGRAPHY

